(2)

Carnegie Mellon University
Software Engineering Institute

Serpent

Overview

System for User Interface Development	Version	Date
	1	May 1991

User's Guide

CMU/SEI-91-UG-1
May 1991

Serpent Overview

User Interface Project

Approved for public release.
Distribution unlimited.

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/AVS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

John S. Herman, Capt, USAF
SEI Joint Program Office

The Software Engineering Institute is sponsored by the U.S. Department of Defense.

This report was funded by the U.S. Department of Defense.

Preface

This document provides an overview of the Serpent system. It is intended for software engineers involved in user interface development and assumes no previous knowledge of Serpent.

Preface

Introduction

Serpent is a user interface management system (UIMS) being developed at the Software Engineering Institute (SEI). Serpent supports the development and implementation of user interfaces, providing an editor to specify the user interface and a runtime system that enables communication between the application and the end user.

What Is a User Interface Management System?

A UIMS is a set of tools for the specification and execution of the user interface portion of the system. A UIMS provides tools for the specification of the static, layout portion of the user interface, for the specification of the dynamic portion, and for the execution of the specifications. A UIMS also separates the user interface portion of a system from the functional portion, allowing for modifications to the user interface with minimal impact on the remainder of the system.

The user interface is a major concern of most computing systems and, generally, is distinct from the concerns of the application. Separating the user interface from the application leads to a three-part division of a software system: the presentation of the user interface, the functionality of the application, and the mapping between the user interface and the application.

The advantages of this division are that:

- It allows modifications of the user interface to be done with minimal modification to the functional portion and vice versa. It does this by isolating the functional portion of the application from the details of the user interface. For example, whether a command is specified through a menu choice or through a textual string is not relevant to the functional portion of an application. Removing these concerns from the functional portion of the application allows the type of interface to be modified dramatically without any modifications to the application.

- It allows the development of tools that are specialized for the design, specification, and execution of the user interface. For example, a layout editor, a dynamic specification language, and a runtime to support them can be included.

What Is Serpent?

Serpent is a UIMS that supports the incremental development of the user interface from prototyping through production and maintenance. It does this by providing an interactive layout editor for prototyping, by integrating the layout editor with a dynamic specification language for production and maintenance, and by having an open architecture so that new user interface functionality can be added during the maintenance phase.

The basic features of Serpent are simple enough for use during the prototyping phase, yet sophisticated enough for use in developing the prototype into an operational system. Serpent is designed to be extensible in the user interface toolkits that can be supported. Hence, a system developed using Serpent can be migrated to new technologies without time-consuming and expensive re-engineering of the application portion.

Serpent consists of:

- A language designed for the specification of user interfaces.
- A language to define the interface between the application and Serpent.
- A transaction processing library.
- An interactive editor for the specification of dialogues and for the construction and previewing of displays.
- Input/output (I/O) technologies.

Serpent Features

Serpent provides many features to address the requirements, development, and maintenance phases of a project. For the requirements phase, Serpent provides a language and an editor to define the user interface. For the development phase, Serpent provides a set of tc ls that simplify the development of the user interface. For the maintenance phase, Serpent allows integration of new technologies as well as the ability to modify the user interface. Specifically, Serpent:

- Provides generality in supporting a wide range of both applications and I/O toolkits through its use of database-like schemas.
- Provides a set of tools that simplify the user interface implementation process.
- Encourages the separation of software systems into user interface and "core" application portions, a separation which will decrease the cost of subsequent modifications to the system.
- Supports rapid prototyping and incremental development of user interfaces.
- Facilitates the integration of new user interface toolkits into the user interface portion of a system.
- Supports both synchronous and asynchronous communication. This allows real-time applications to satisfy timing constraints without waiting for user input.

Serpent Documentation

The following documents provide information about the Serpent system.

Serpent Overview
Introduces the Serpent system.

Serpent: System Guide
Describes installation procedures, specific input/output file descriptions for intermediate sites and other information necessary to set up a Serpent application.

Serpent: Saddle User's Guide
Describes the language that is used to specify interfaces between an application and Serpent.

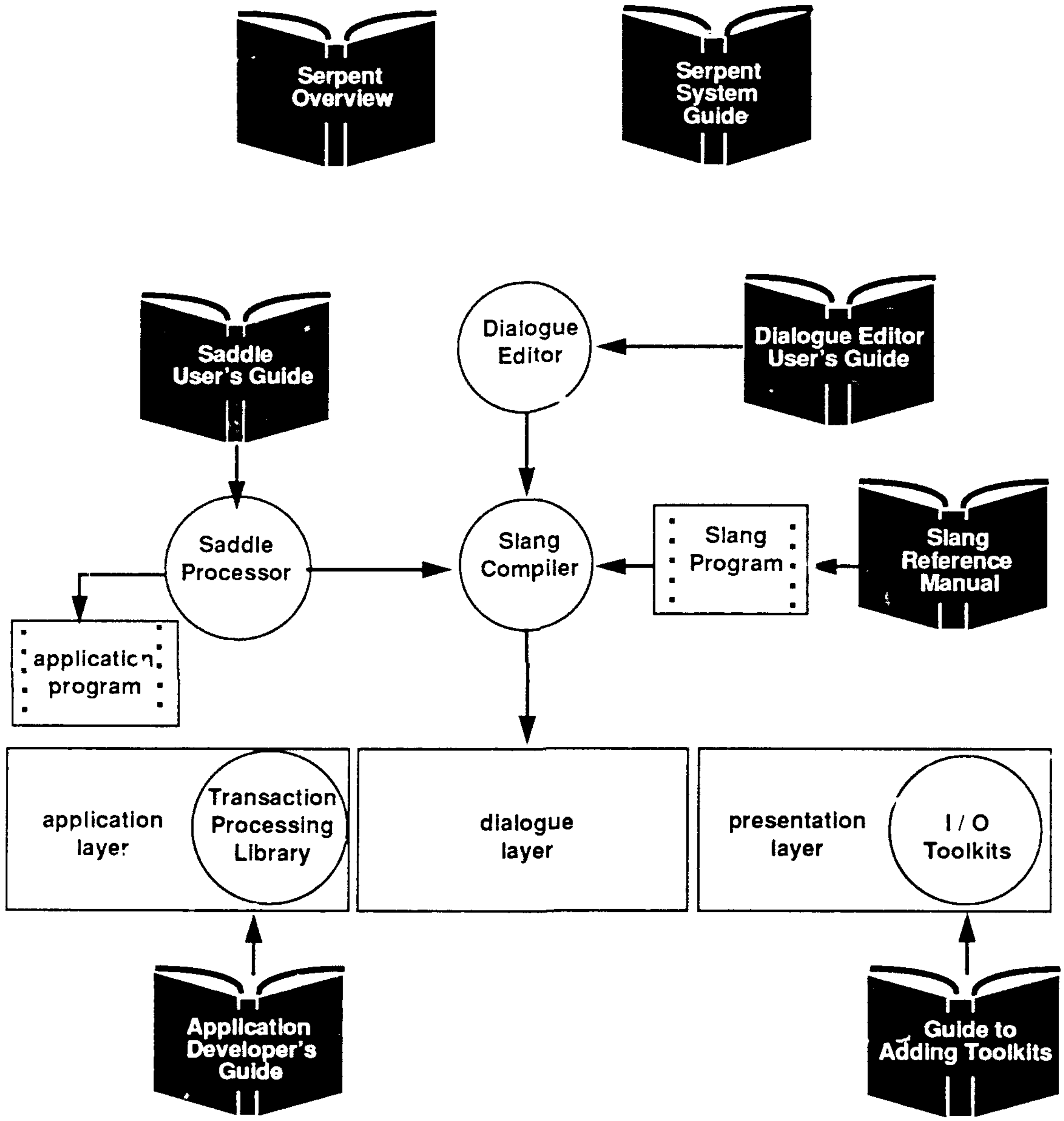
Serpent: Dialogue Editor User's Guide
Describes how to use the editor to develop and maintain a dialogue.

Serpent: Slang Reference Manual
Provides a complete reference to Slang, the language used to specify a dialogue.

Serpent: C Application Developer's Guide
Serpent: Ada Application Developer's Guide
Describe how the application interacts with Serpent. These guides describe the runtime interface library, which includes routines that manage such functions as timing, notification of actions, and identification of specific instances of the data.

Serpent: Guide to Adding Toolkits
Describes how to add user interface toolkits, such as various Xt-based widget sets, to Serpent or to an existing Serpent application. Currently, Serpent includes bindings to the Athena Widget Set and the Motif Widget Set.

The following figure shows Serpent documentation in relation to the Serpent system:



Serpent Documents

Introduction

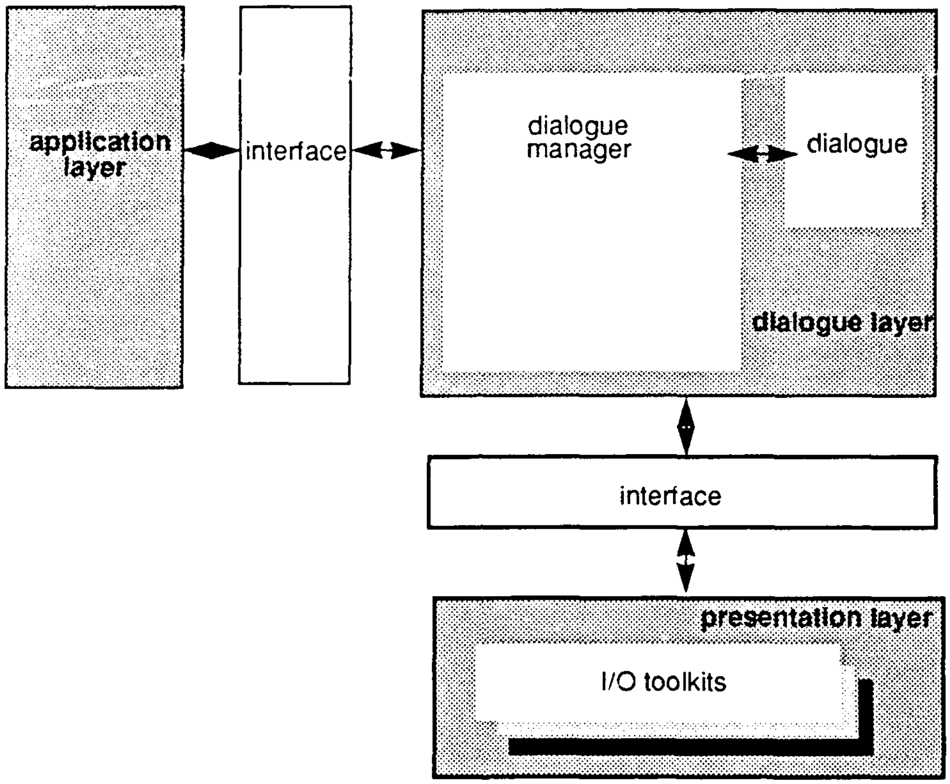
System Description

Serpent is based on the UIMS architecture defined by the Seeheim working group on graphical interfaces. The architecture consists of three layers:

1. The presentation layer is responsible for layout and device issues.
2. The dialogue layer specifies the presentation of application information and user interactions
3. The application layer provides the functionality for the system under development.

System Description

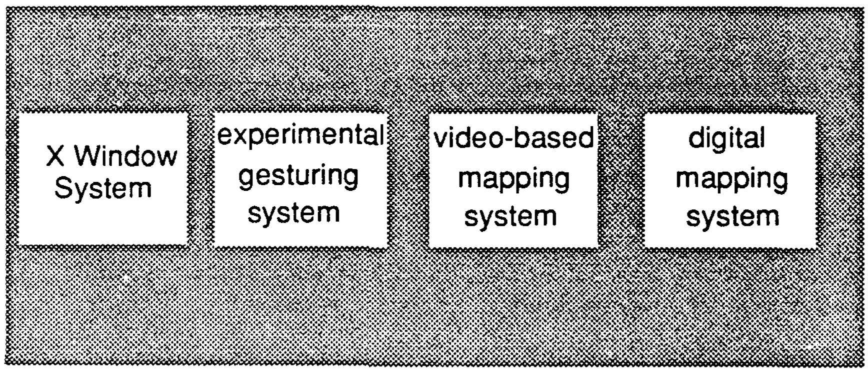
The architecture is intended to encourage the proper separation of functionality between the application and the user interface portions of a software system and to allow for the development of specialized tools to support the different layers. The three different layers of the standard architecture provide differing levels of control over user input and system output.



Serpent Architecture

Presentation Layer

The presentation layer controls the end user interactions and generates low-level feedback. As illustrated in the following figure, the presentation layer consists of various I/O toolkits that have been incorporated into Serpent. A standard interface is used that simplifies the addition of new toolkits. Each I/O toolkit defines a collection of interaction objects visible to the end user. The interaction objects that exist at this level depend upon which I/O toolkits have been integrated into Serpent at a particular installation. Currently, Serpent supports the X Window System, Version 11, with the Athena Widget set and the OSF Motif Widget set.



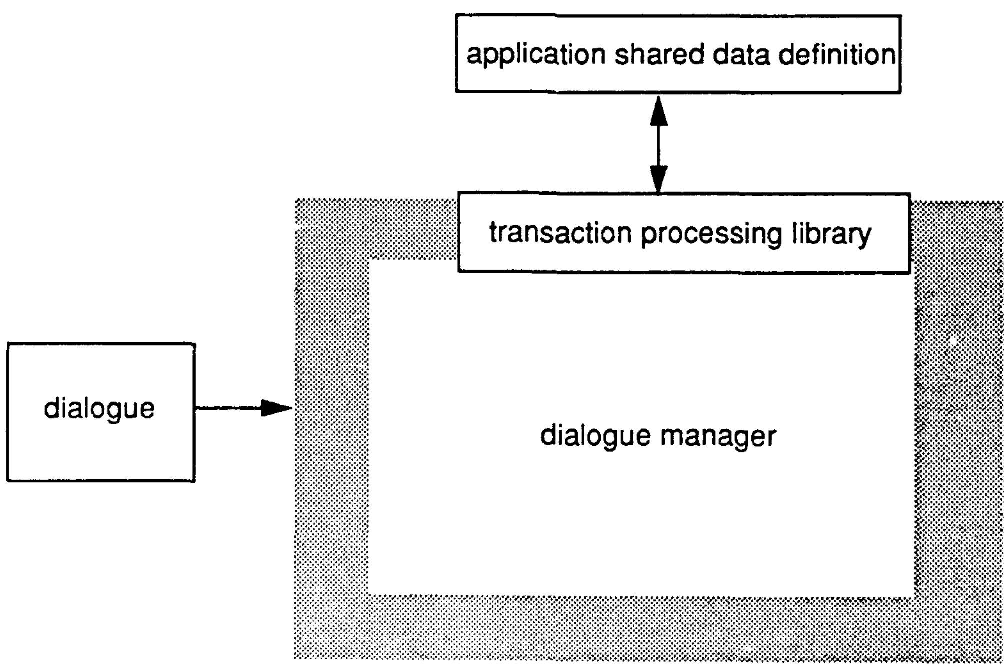
Presentation Layer

Dialogue Layer

The dialogue layer specifies the user interface and provides the mapping between the presentation and application layers.

The data that is passed between the application layer and the dialogue layer is referred to as *application shared data.* The *application shared data definition* provides the format of the data, while the application shared data represents the actual value of the data.

The dialogue layer determines which information is currently available to the end user and specifies the form that the presentation will take as previously defined by the dialogue specifier (individual responsible for creating the dialogue). The dialogue layer acts as a traffic manager for communications between the application and I/O toolkits. The presentation layer manages the presentation; the dialogue layer tells the presentation layer what to do. For example, the presentation layer will display a button that the end user can select; the dialogue layer will tell the presentation layer the position and the contents of the button, and will respond when the button is selected.

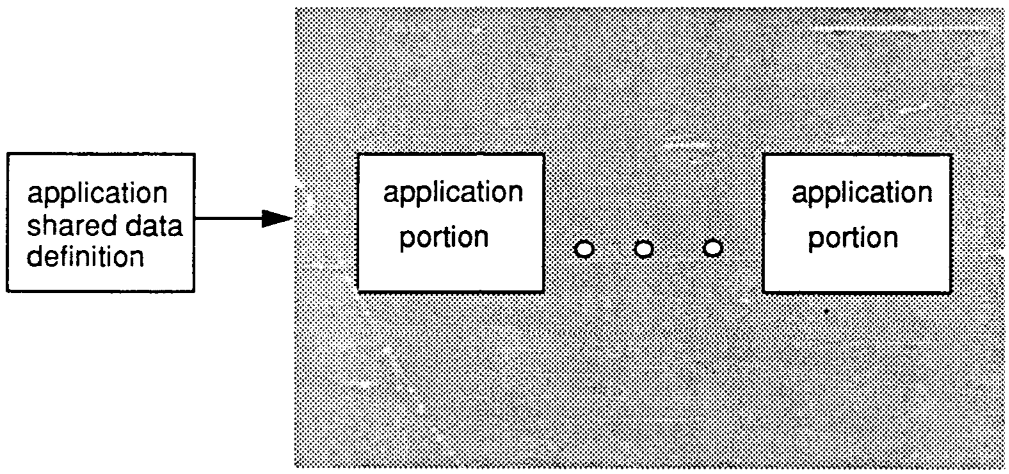


Dialogue Layer

Application Layer

The application layer performs those functions that are specific to the application. Since the other two layers are designed to take care of all the user interface details, the application should be written to be presentation-independent: there should be no dependency upon a specific I/O toolkit. While the application developer should be aware that there is an end user and should provide the end user with information, that information should be presented in application terms.

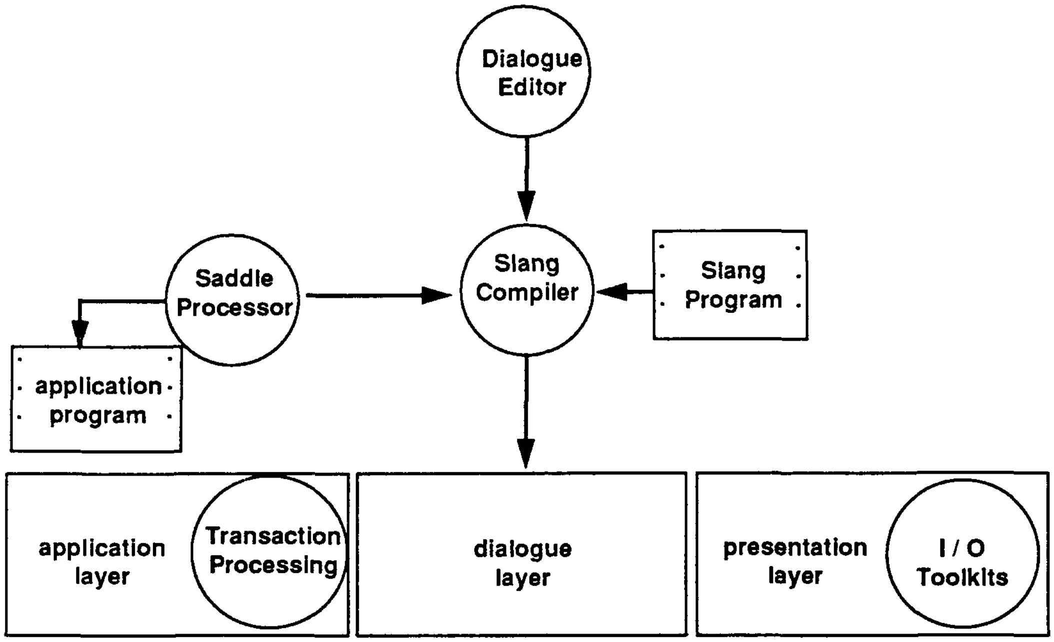
The application layer is not part of the Serpent system. Serpent currently supports applications developed in C or Ada.



Application Layer

Serpent Components

The components of Serpent are highlighted in the illustration below. A short summary of each of the components follows.



Serpent Components

Dialogue editor - Interactive editor used to specify a dialogue. The editor contains manipulation capabilities to construct and preview a dialogue.

I/O toolkits - Components of the presentation layer. Included are both the toolkits that have been integrated into Serpent and the integration code itself.

Saddle - Database-like schema language used to define the interface between the application layer and the dialogue layer.

Slang - Language used to specify the dialogue. This language is compiled into the dialogue layer of the Serpent architecture.

Transaction processing library - Library of procedures linked to the application layer to provide access to Serpent.

Using Serpent for Prototyping

Serpent is designed to allow incremental development of the user interface from prototyping through production. Serpent can be used either with or without an application layer.

Simple prototyping is the construction of displays with a fixed collection of objects on the screen. The visual aspects of these objects are best specified with the dialogue editor and the active aspects are best specified using the Serpent textual dialogue description language, Slang. This chapter presents the concepts that underlie the dialogue description.

In Serpent terms, a simple prototype has no application layer.

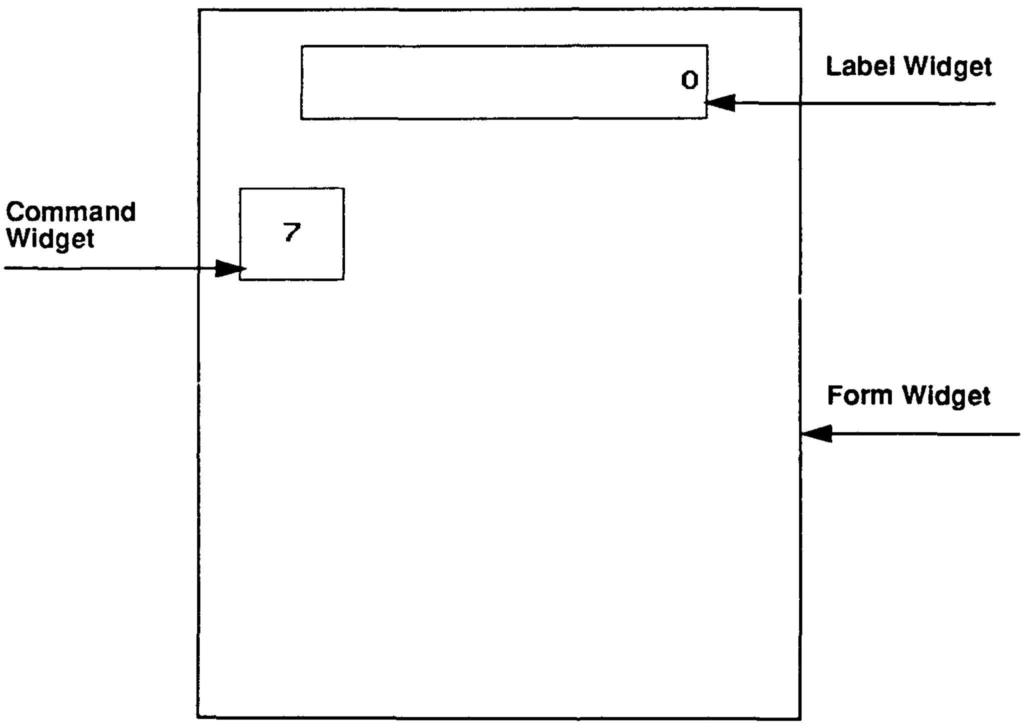
Objects

Objects are the means by which an end user visualizes and interacts with the presentation layer. Examples of objects are instances of Athena command widgets. Objects are defined in the dialogue, which is represented in Slang. Each presentation technology defines a set of primitive object types that may be used in a dialogue. For example, the Athena widget set includes object types such as: forms, command buttons, and textual input. Each object type has a collection of attributes that define its presentation, as well as methods that determine the high-level interactions that the end user can have with the object. Objects are specified in a Slang program by listing the objects to be created and the attribute values to be assigned to each occurrence of the object. Objects created using the dialogue editor are maintained internally in Slang.

Object Types

The objects used in the Sepent program examples at the end of this manual are X Toolkit Athena Widgets. A complete list of these objects and their attributes available through Serpent can be found in Appendix B in *Serpent: Slang Reference Manual.*

The examples use three types of Athena Widget objects—form widget, label widget, and command widget.



Object Types

The form widget defines an area on the screen on which other objects can be placed. If the form is moved, all of the objects on it are also moved. The required attributes necessary for a form are its height and width. All of the other attributes contain default values. For example, in constructing a calculator, the form widget provides the outline for the calculator.

A second type of object is a label widget. This type of widget is used for displaying text that the end user cannot select. (For example, the user cannot select or edit the display of a calculator.) The attributes for the label widget are described in the following table.

Label Widget	Description
parent	The form widget that acts as the base for the widget.
x, y	The position of the upper left corner of the widget with respect to the upper left corner of the parent in pixels.
height, width	The height and width of the label widget in pixels.
label	The text displayed within the label.

Label and Command Widget Attributes

A third type of object is a command widget. This type of widget is used to display text denoting possible commands that the end user can select. The attributes for the command widget are the same as the attributes for the label widget.

A fourth type of object, the text widget, allows an end user to input text into a field. The text is maintained in the widget within the presentation layer until the dialogue explicitly asks for the text. Dialogues that use text widgets tend to be somewhat complicated because of this feature. The use of a text widget is demonstrated in Example 4 of the Serpent programs included at the end of this manual.

Enumerating Objects in Slang

The Slang code for displaying the digit 7 on a calculator is shown below. Objects are enumerated in Slang by name and type. Within each object, the attributes are enumerated and given a value.

```
digit_7: XawCommand: {
 Attributes:
    parent: main_background;
    horizDistance: 20;
    vertDistance: 100;
    height: 50;
    width: 50;
    label: 7;
    justify: 2;
}
```

Enumerating C ɔjects in Slang

Methods

Some of the object types in the presentation layer allow end user interaction. These end user interactions are handled in a Slang program by defining a *method* for that particular object. The command widget allows the end user to select the widget. In the presentation layer, the selection is converted into a `notify` method for that object, which causes the statements in the method portion of the object definition to be executed. Example 2 in the example programs at the end of this manual demonstrates the concept of a method.

Variables

A Slang program can have variables that are used for intermediate calculations. Attribute definitions and methods can use a collection of statements that give very flexible arithmetic calculations. The number of variables used depends on the number of calculations.

In the following example, the variable `display` holds the value to be shown in the display portion of the calculator. When the method for the command widget is executed, the value of the variable is modified.

```
display_widget: XawLabel {
 Attributes:
    parent: main_background;
    horizDistance: 50;
    vertdistance: 20;
    height: 40;
    label: display;
    justify: 2;
}

digit7: XawCommand {
 Attributes:
    parent: main_background;
    horizDistance: 20;
    vertDistance: 100;
    height: 50;
    width: 50;
    label: 7;
```

Use of a Variable

Dependencies Between Variables and Attributes

In Slang, dependencies between calculations are automatically enforced. This is a very important and powerful feature. It frees an interface specifier from the implications of modifying a variable. The interface specifier has to modify only the variable; Slang automatically updates all of the attributes and variables that depend upon that variable.

In the calculator example, the text to be output to the display of the calculator is specified as the value of the variable `display`. When the value of the variable is modified, the output automatically changes. The value is initially set to 0 and when, for example, the digit labeled 7 is selected, the value of the variable is changed to 7. Changing the value of `display` automatically causes the attribute `label` in `display_widget` to be modified and that modification to be communicated to the presentation layer and, consequently, to the end user.

Specifying a prototype interface requires only:

1. Specifying the objects to be displayed and their positions. This is usually done with the layout portion of the dialogue editor.
2. Deciding which attributes might be changed through end user interactions and use of variables (or arithmetic functions with variables) to set the values of these attributes. This is usually done through the textual portion of the dialogue editor.
3. Specifying the actions to occur on end user interaction with the displayed objects. This is usually done through the textual portion of the dialogue editor.
4. Iterating through the layout and the dynamics given in steps 1-3.

Serpent takes all the steps necessary to update the displays when one of the variables is modified. *Serpent: Slang Reference Manual* discusses this concept in more detail.

How to Control the Existence of Objects

Serpent features provide for the presentation of objects, the handling of end user interactions on those objects, and the automatic enforcement of dependencies. Serpent has features that also allow the creation and deletion of objects, both individually and in logical groups.

View Controllers

The mechanism by which objects are grouped, or collected, is a *view controller*. A view controller performs two main functions:

1. Mapping specific data in the application to display objects with which the end user can interact.
2. Controlling the existence of groups of objects.

A dialogue is specified in terms of *view controller templates*. A template maintains a watch on application shared data for specific conditions. When data that satisfies a view controller template is placed into application shared data, a view controller is created.

A Slang program specifies a view controller template with which a group of objects is associated. When a view controller is instantiated (created) from the template, the associated objects are created and communicated to the presentation layer. The presentation layer makes the objects visible to the end user.

For example, suppose a menu bar with a list of options is displayed. When one of the options is chosen, a pull down menu appears. Each pull down menu is implemented in a distinct view controller. Example 3 in the example programs included at the end of this manual contains a Slang program that implements a menu bar in this manner.

Creation Conditions

Each view controller template has a *creation condition* that specifies the condition under which a template is instantiated. The creation condition can be any boolean expression. When the boolean expression becomes true, a view controller and its associated objects are instantiated from the template. When the boolean condition becomes false, the view controller and its associated objects are deleted.

Nesting View Controllers

It is possible to specify that one view controller template be nested within another view controller template. This nesting extends to the view controllers created from the templates: a nested view controller inherits the data from its predecessor. When a view controller is deleted, all of its nested view controllers are also deleted.

Control Within Dialogues

The order in which the view controllers are created depends on the data that the application places into shared data and on the actions of the end user. A subdialogue is a collection of view controllers that perform one particular task. It is possible to have multiple subdialogues within a dialogue. There are no *a priori* timing constraints on the execution order for the subdialogues.

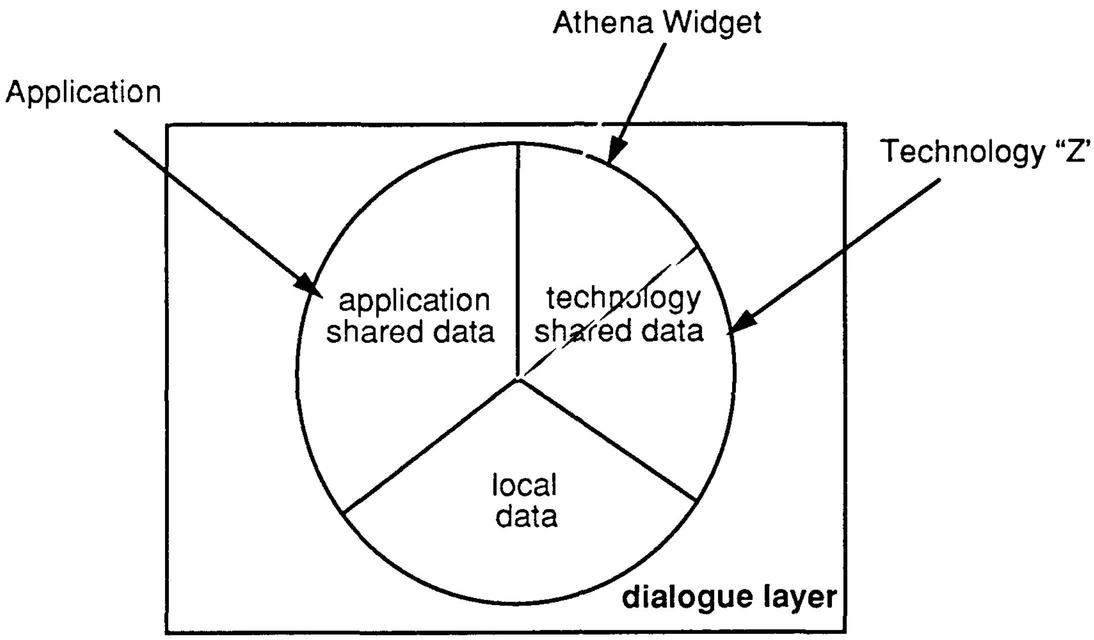
Actions of the dialogue are determined by the actions of the application and those of the end user. Multiple subdialogues can be active simultaneously. Within Serpent, view controller creation and deletion take place only in response to end user and application actions. In particular, several subdialogues may be carried on in parallel. This parallel execution of subdialogues represents the power of the production model used in Serpent.

Using Serpent with an Application

The full power of Serpent is demonstrated when used with an application. When data is shared, it is possible to have multiple instances of a particular view controller template. This is one of the powerful, although somewhat complicated, features of Serpent. Shared data provides the interface between Serpent and the application.

Shared Data

From the application developer's perspective, Serpent behaves like an active database management system. The data that is managed by Serpent (except for variables declared within view controller templates) is called *shared data*. Data is manipulated within the shared database by the application, the presentation layer (usually in response to end user actions), or the dialogue layer (in response to actions of the dialogue). The shared database is segmented into sections associated with the application, presentation, and dialogue layers.



Serpent Shared Database

The section associated with the application (*application shared data*) is accessible only from the application layer and the dialogue layer. The section associated with the presentation layer (*toolkit shared data*) is accessible only from the presentation layer and the dialogue layer.

From this perspective, the actions of the dialogue layer are to retrieve data from application shared data and transform that data into toolkit shared data and, also, to retrieve data from toolkit shared data and transform it into application shared data. This simplified view ignores the fact that the dialogue could transform some application or toolkit shared data into other application shared data. It also ignores the fact that toolkit shared data is actually segmented into a collection of shared data specific to the I/O toolkits that have been integrated into Serpent.

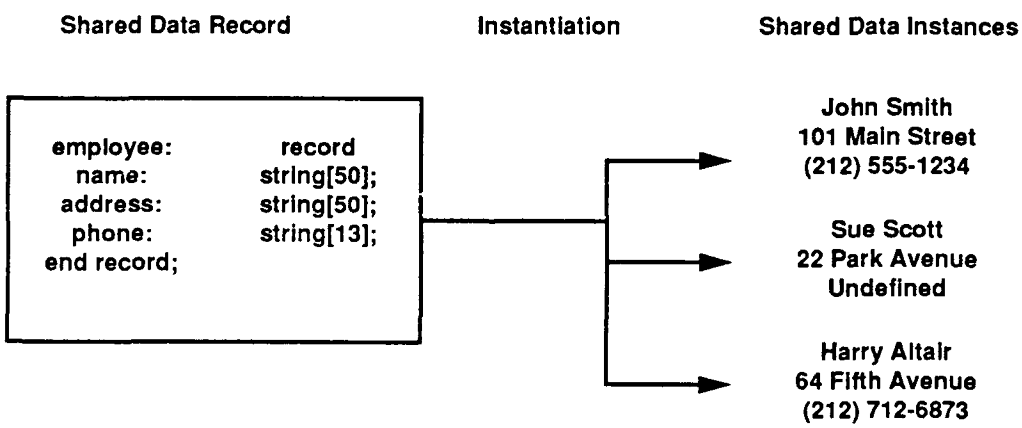
Serpent combines communications between the application and dialogue layers into transactions to protect against collisions of data between the two layers.

Shared Data Definition File

The type and structure of data that is maintained in the shared database is defined externally in a *shared data definition file*. This corresponds to the database concept of schema. A shared data definition file is created once for each application and once for each technology that is integrated into Serpent. Creation of this file is necessary before an application can use Serpent. The language used to define the shared data definition is called Saddle. Saddle is more completely described in *Serpent: Saddle User's Guide.*

A shared data definition file consists primarily of aggregate data structures (records). These records represent a structure that is instantiated at runtime. The term *shared data element* is used to refer to the aggregate structure.

The notion that instances exist of shared data records is important. Examples of instances of an employee record within a shared database are shown below. The shared data record (depicted in the following figure) is specified in the shared data definition file. At runtime, three employee instances were added to the shared database. That is, the record was instantiated three times. Each shared data instance is identified by a unique name, known as its ID.



Description Mechanism

A file external to the dialogue and the application contains the description of the shared data structure. The description is in Saddle, a shared data description language tailored especially for Serpent. The file is processed to produce a language-specific description of shared data. Processors exist for Ada and C. If the application is written in C, the processor will generate structure definitions that can be included in the application program. If the application is written in Ada, the processor will generate package specifications. (For more information about Saddle, refer to *Serpent: Saddle User's Guide.*)

The Application Perspective

Serpent provides an active database view to the application. The application can add, modify, or delete data from the shared database. Information provided to Serpent by the application is available for presentation to the end user. The application has no knowledge of the presentation media or user interface styles used to present the information.

When the end user, through the dialogue, adds, modifies, or deletes data from the application shared database, the application is informed. The shared database mechanism also is used to communicate commands from the dialogue to the application. For example, instructing the application to add a new employee could be accomplished by having a menu with an "add" selection, or by using a command-style syntax. In the first case, the dialogue would inform the application that an "add" was selected, and in the second, the dialogue would parse the command syntax and inform the application that an "add" was to be accomplished. The application is kept ignorant of the mechanism by which the command is entered.

Using Slang with an Application

At this point, the concepts necessary to use Serpent with an application have all been introduced:

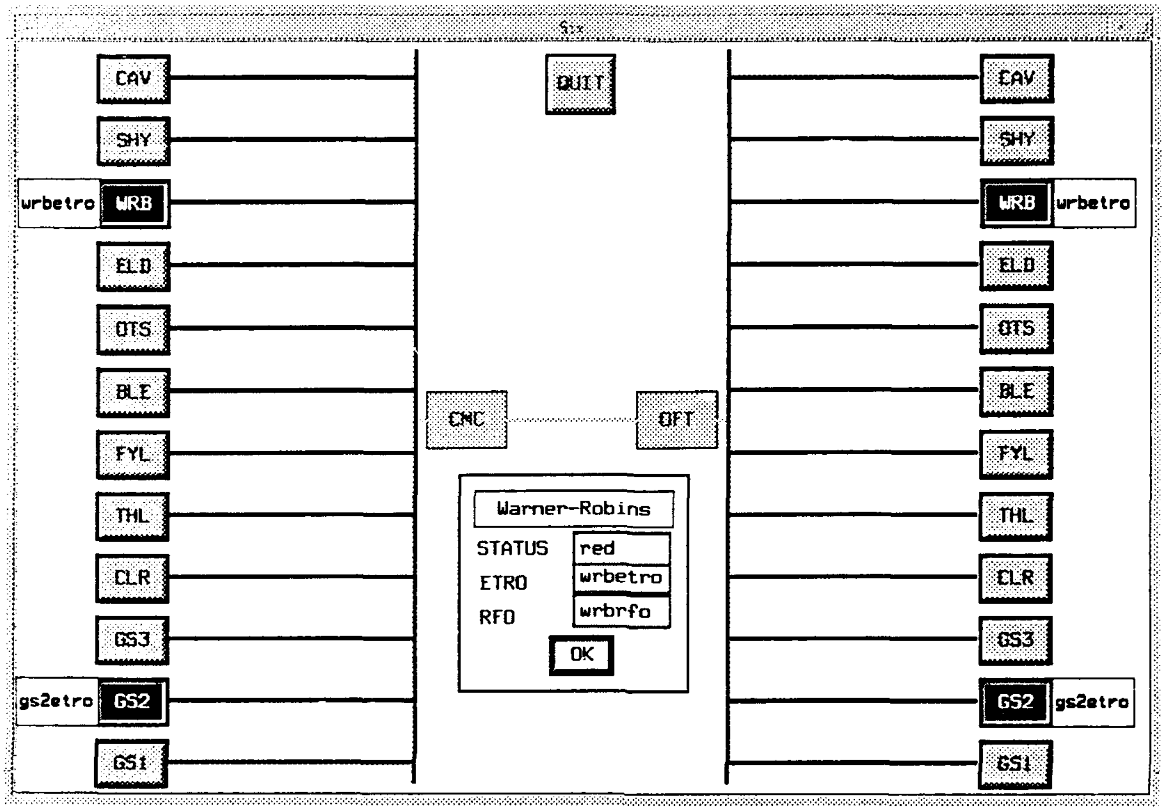
- Instances of application shared data elements as individual application objects.
- Objects as both presentation and interaction mechanisms.
- View controllers as logical collections of objects.

The application creates the instances of application shared data through the use of the Serpent library. In the dialogue, view controller templates are defined. Each template has a creation condition that specifies under what condition the view controller is to be created. If that condition includes a reference to application shared data, then a new view controller is created whenever an application shared data instance is placed into shared data.

A view controller template can be instantiated for each instance of a particular shared data element. Thus, a single view controller template could be used to display data for each of the employees represented in the shared data instantiation illustration. Each instance of employee data would cause a view controller instance to be created, and each view controller would have its associated collection of objects presented. Some of the attributes in the objects would depend upon the components of the employee data record, and the same dependency conditions that are enforced for local variables would also be enforced for application shared data.

In other words, for a given employee, that employee represents one instance in shared data, and that instance is bound to a particular view controller. The view controller has an object whose text is determined by the shared data component `employee.address`. If the value of an instance of `employee.address` changes, the object being displayed with that address will automatically be updated by Serpent.

Application Example



Sensor Site

This figure illustrates the display an end user sees for a command and control application. The rectangular boxes on the right and left sides (e.g., GS1 and GS2) represent sensor sites that detect information. The boxes in the middle (CMC and OFT) represent correlation centers where the information from all of the sensors is collected. Each sensor site sends its information to both correlation centers, which accounts for the repetition of the sensor site boxes on both the right and left sides of the display. The lines represent the communication path between a particular sensor site and a correlation center.

The end user may select one of the sensor sites, and a detail window will appear giving more status information about the site. This detail window may be used to modify the estimated time to return to operation (ETRO), the status, and the reason for failure (RFO). The figure shows the result of selecting the WRB sensor.

Dialogue Editor

The dialogue editor is an interactive tool used to specify a dialogue. A dialogue consists of a static layout portion and a dynamic behavioral portion, and the editor is used to specify both portions. The output of the editor is a Slang program that can then be compiled and executed in the same fashion as any other Slang program.

The layout portion of a dialogue is the placement of objects on the screen. If the position and size of the objects are constant then the layout portion can be specified directly prior to execution. If the position or size varies, the layout specifies the initial position of the objects. The portion of the editor that is used to specify the layout is called the *layout editor*. The editor has the capability to create objects through the layout editor and modify their attributes.

The dynamic portion of a dialogue controls the creation and deletion of objects and the modification of attributes to change position or appearance. The portion of the editor that is used to specify dynamics is called the *structure editor*.

When an object is created within the layout editor, a template for that object is created and is accessible through the structure editor. Thus, a particular attribute or collection of attributes can be made dynamic.

It is possible to instantiate all of the objects that exist in a view controller. Thus, an iterative dialogue design method that calls for laying out a portion of the dialogue, adding and testing some of the dynamic portions, and so on is possible with the dialogue editor.

Finally, the dialogue editor is itself an application that uses Serpent. That is, the application layer is responsible for saving and restoring dialogues, for generating the Slang program, and for other non-visible types of activities. Serpent is responsible for presentation of menus, interpretation of selections, presentation of the visual representation of the dialogue, and other visible and non-functional activities.

Visual Presentation of Displays

Each view controller template contains a collection of objects. The objects associated with a particular view controller can be previewed at specification time without executing the Slang program. Thus, the dialogue specifier is able to see the size and relationship of the objects more quickly. The dialogue specifier can modify the size and position of an object either directly with a mouse or indirectly through modification of the attributes of the object.

The dialogue specifier selects the objects to preview by view controller. Because a particular display may represent a collection of view controllers, the time at which the collection is to be previewed can be specified. This allows the dialogue specifier to get some idea of the effect of sequencing through displays.

The dialogue specifier can also use the preview mechanism to create and delete objects. Objects can be created and left indeterminate with respect to particular view controllers. At a later point in the evolution of a dialogue, objects can be assigned to particular view controllers and moved from one view controller to another.

Structure Editor

The editor provides both a structured presentation of the Slang language and a preview of the constructed displays. The structured presentation of the language simplifies the syntactic specification of a Slang program. The preview of constructed displays allows the individual responsible for creating the dialogue (the dialogue specifier) to see the static structure of the displays without the necessity of compiling and executing the Slang program.

A Slang program consists of syntactic superstructure (e.g., identifying something as a creation condition) and procedural code to perform some action (e.g., calculating the attribute of an object). The editor allows the dialogue specifier to focus on the procedural aspects of a dialogue. For example, a creation condition is specified in Slang by the syntax:

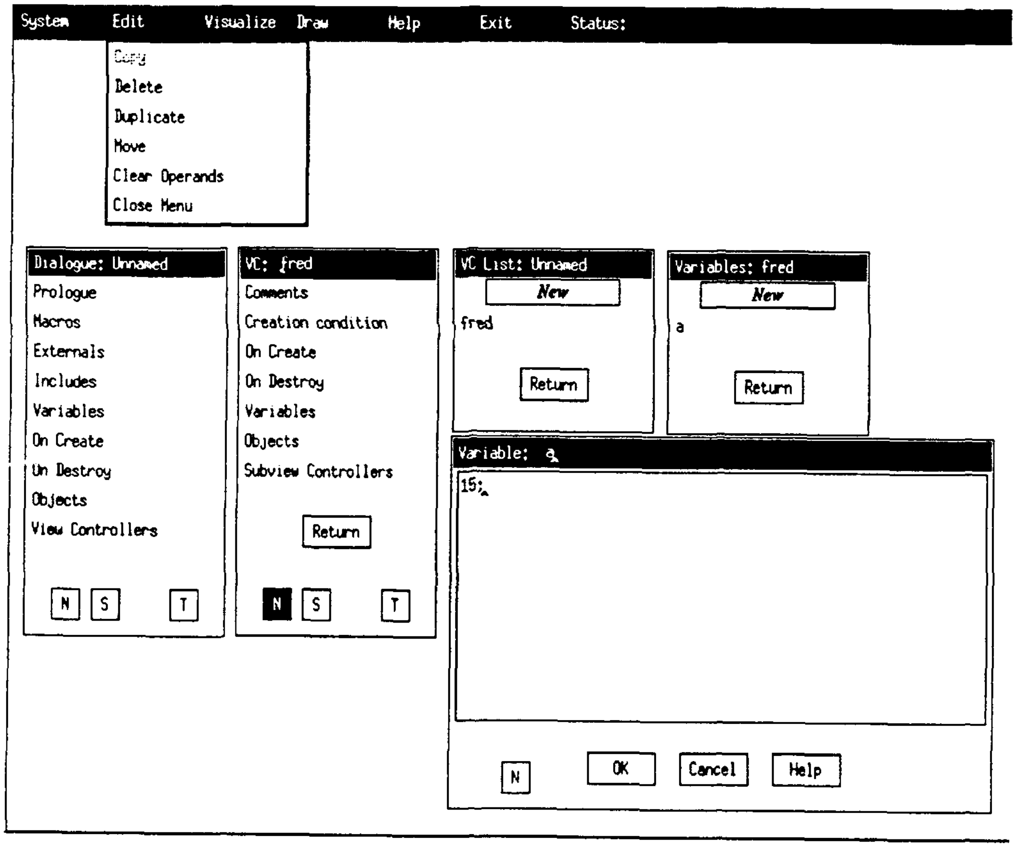
Creation Condition: (Actual Condition)

Dialogue Editor

Using the editor, the dialogue specifier would select creation condition on a menu associated with a view controller and be able to edit the actual condition. When the editor transforms the dialogue into a Slang program, the editor creates the syntax:

Creation Condition: (...)

and places the actual condition inside the parentheses. Thus, the editor eliminates the need to be concerned with the detailed syntax of the Slang program. A dialogue specifier has access through visual means to all of the components of a dialogue; that is, view controllers, objects, attributes of objects, creation conditions, and variable declarations. The logical portion of a dialogue is specified textually.



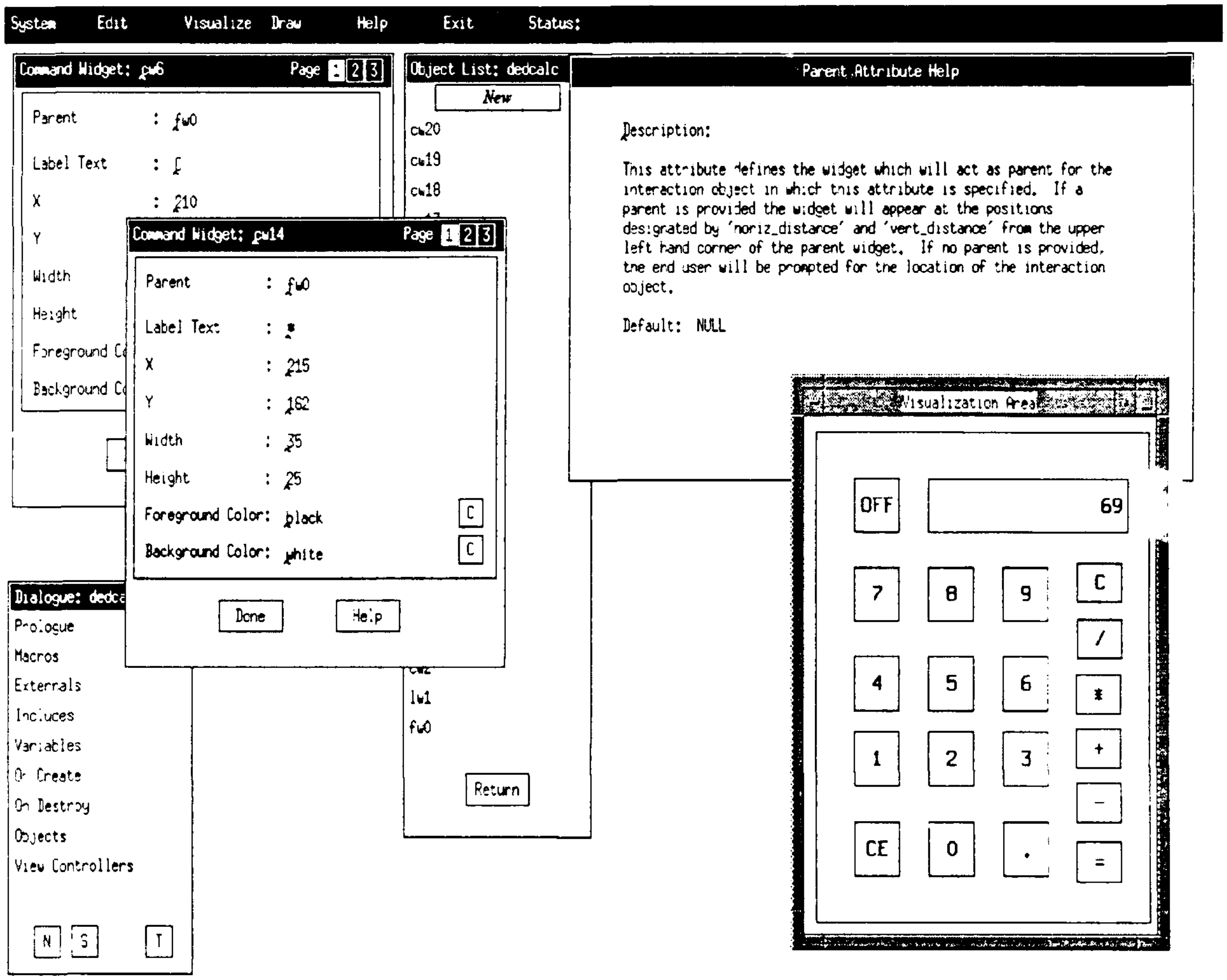
Editing the Dialogue

Thus, two approaches to specify the dialogue are:

1. A structured, top-down definition of the dialogue, defining view controllers and the objects contained within them.
2. A bottom-up definition of the dialogue, defining the objects and subsequently collecting them into view controllers.

When all of the attributes of the objects are known at the time of dialogue specification (that is, when the attributes are constants,) Serpent knows how to display the objects. When some of the attributes depend upon either local variables or shared data, previewing the objects becomes more complicated. A specific value needs to be assigned to those attributes at specification time so that Serpent will know how to display the objects.

The editor sees the preview values of attributes differently from their calculation values. If the attribute is a constant, the values are identical and a modification to one view of the attribute is considered to be a modification of both views. If the attribute must be calculated at execution time, the values are different.



Editing the Layout

Dialogue Editor

Integrating New Input/Output Toolkits

Serpent is designed to allow the integration of new input/output toolkits into the presentation layer. Toolkits are integrated into Serpent through mechanisms that are similar to those used by applications.

The following are steps for integrating a new technology into Serpent:

1. Decide upon the objects to be presented to the end user and their attributes.
2. Define a shared data definition that reflects the objects defined in Step 1.
3. Develop a layer of software that translates between the objects defined in Step 1 and the internal representations of the technology.
4. Register the technology with the dialogue editor.

Except for following communication conventions, a technology is integrated into Serpent using the same mechanism and techniques as an application written using Serpent. Integrating new I/O toolkits is described completely in *Serpent: Guide to Adding Toolkits.*

Toolkits based on Xt (the Intrinsics of the X Toolkit) are more easily integrated into Serpent. A special purpose language for describing such toolkits exists, and all of the mechanisms exist that are necessary to include such toolkits into the presentation layer and the layout portion of the editor. These mechanisms are also described completely in *Serpent: Guide to Adding Toolkits.*

Glossary

application data

Data provided to Serpent by the application.

application developer

The individual responsible for developing the application.

application layer

The layer that provides the functionality for the system being developed.

application shared data

The section of the shared database associated with the application. This data acts as the interface between the functional portion of the application system and Serpent.

attribute

A characteristic of an interaction object that is specified by the dialogue.

creation condition

The specified condition under which a view controller is created.

dialogue

A specification of the user interface.

dialogue layer

The layer that implements the dialogue between the user and the application.

dialogue manager

The part of Serpent that executes the dialogue.

dialogue specifier

The individual responsible for creating the dialogues.

end user

The individual who uses the system developed with Serpent.

ID

A unique internal name of a shared data element.

Glossary

input/output (I/O) toolkits

Hardware/software systems that perform some level of generalized interaction with the end user.

interaction objects

Objects by which the user and application communicate.

interface specifier

The individual responsible for developing the user interface Serpent.

layout editor

The layout editor specifies the position of objects on the screen.

method

A way of handling end user interactions in the dialogue by specifying actions to be performed for specific end user generated events.

object

The means by which an end user interacts with the presentation layer.

presentation independent

A presentation that is independent of the user interface of the system.

presentation layer

The layer responsible for layout and device issues.

preview of objects

A Serpent feature that provides the dialogue specifier a view of objects at specification time without the Slang program's having to be executed.

record

A collection of simple objects.

Saddle

A special-purpose data declaration language used by Serpent to describe the structure of the shared data.

schema

A description used by the interface to define the type and structure of data shared between the major system components.

Glossary

Serpent

A user interface management system being developed by the Software Engineering Institute. Serpent supports the separation of a computing system into an application portion and a presentation portion.

shared data

The data (except for variables declared within view controller templates) that is managed by Serpent.

shared database

All data managed by Serpent. The database consists of application data, presentation data, and global dialogue data.

shared data element

A shared data element is a description of a particular portion of shared data. It has attributes called components. Instances of elements are called IDs, occur at runtime, and are managed by the runtime interface library.

shared data definition file

A description in Saddle of the shared data.

simple prototype

The construction of displays without the application for which the interface is being designed.

Slang

A fourth-generation language for dialogue specification created specifically for Serpent.

structure editor

The structure editor controls the dynamic portion of a dialogue which affects the creation and deletion of objects and the modification of attributes to change position or appearance.

subdialogue

A collection of view controllers that perform one particular task.

toolkit shared data

Toolkit shared data that is shared between the dialogue manager layer and the presentation layer.

technology integrator

The individual responsible for integrating the selected I/O technology into the Serpent system.

Glossary

view controller

A view controller is an instance of a view controller template. It provides a mechanism for grouping objects according to visibility conditions. A view controller performs two main functions: controlling the existence of groups of objects; mapping specific data in the application to display objects with which the end user can interact.

view controller template

A dialogue is specified in terms of view controller templates. A template maintains a watch on application shared data for specific conditions. When data that satisfies a view controller template is placed into application shared data, a view controller is created.

widget

A name for interaction objects used with the X Window System.

Example Programs

Example 1: Creating Widgets

In this example three components of a screen calculator are created in Slang:

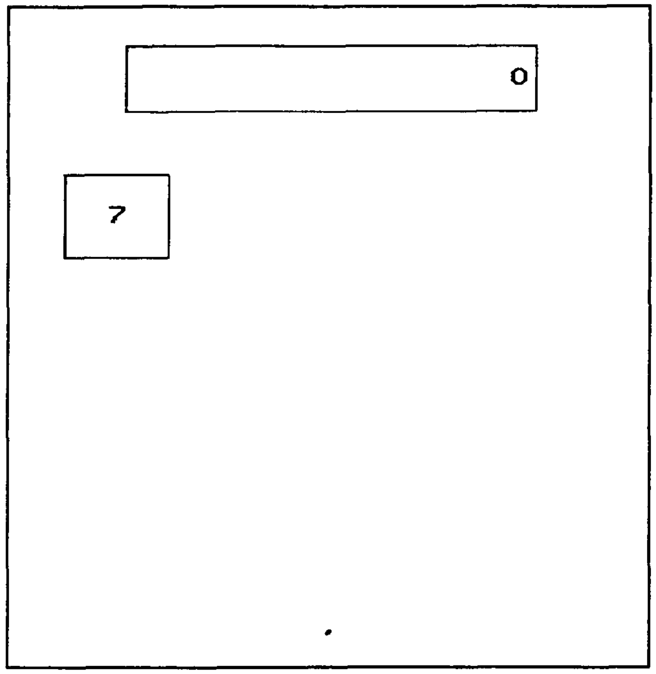
- Form widget, which acts as the outside boundary of the calculator.
- Label widget, which acts as the display portion of the calculator.
- Command widget, which acts as the numeric button 7 on the calculator.

The end user cannot control any of the actions in this example.

In the first line, the `#include "sat.ill"` informs Serpent that this program will be used with the X Window and Toolkit technology. The ||| is a special Slang delimiter.

```
#include "sat.ill"

|||

Objects:

/*
** Object outside boundary of calculator.
*/


main_background: XawBboard {
 Attributes:
    height: 400;
    width: 300;
}

/* object: display on top of calculator.
** For this example, the value will be 0 and will not
**be changed.
*/
```

```
display_widget: XawLabel {
 Attributes:
    parent: main_background;
    horizDistance: 50;
    vertDistance: 20;
    height: 40;
    width: 200;
    label: 0;
    justify: 2;
}
/* object: display digit 7.
** For this example, selecting digit does nothing.
*/
digit7: XawCommand {
 Attributes:
    parent: main_background;
    horizDistance: 20;
    vertDistance: 100;
    height: 50;
    width: 50;
    label: 7;
}
```



Calculator – Creating Widgets

Example 2: Invoking a Method

In this example more of the calculator is displayed, in particular, a second digit and a button that clears the display area. The digit buttons have methods that execute statements when the digit buttons are selected. The clear button has a method that resets the display when selected.

The variable `display` stores the value to be shown in the display portion of the calculator. When the method for any of the command widgets is executed, the value of the `display` variable is modified. The value is initially set to 0 and when, for example, the digit labeled `7` is selected, the value of the variable is set to 7. This automatically causes the attribute `label` to be modified and that modification to be communicated to the presentation layer, and consequently, displayed to the end user.

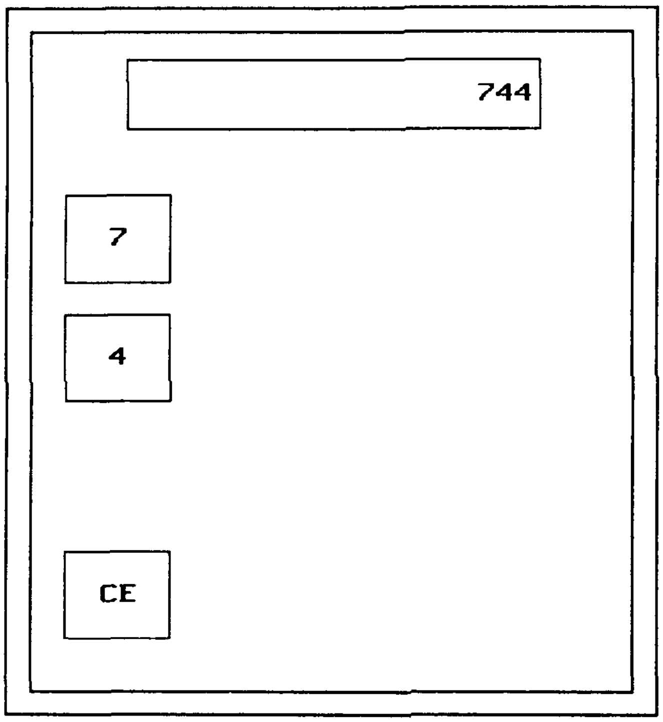
Example Programs

```
#include "sat.ill"

|||

Variables:
/*
** The variable carries the value to be presented in the
** display
*/
 display: 0;
Objects:
/*
Object: outside boundary of calculator.
*/
main_background: XawBboard {
 Attributes:
   height: 400;
   width: 300;
}
/*
** Object: display on top of calculator. For this
** example, the value is given by the variable display.
** It is changed by pushing the digit or clear button.
*/
display_widget: XawLabel {
 Attributes:
    parent: main_background;
    horizDistance: 50;
    vertDistance: 20;
    height: 40;
    width: 200;
    label: display;
    justify: 2;
}
/* Object: display digit 7.
** For this example, selecting digit adds 7 to
** the end of the display.
*/
digit7: XawCommand {
 Attributes:
    parent: main_background;
    horizDistance: 20;
```

```
      vertDistance: 100;
      height: 50;
      width: 50;
      label: 7;
   Methods:
      notify:{
        display := 10 * display + 7;
      }
}
/*
** Object: display digit 4. For this example, selecting
** digit adds 4 to the end of the display.
*/
digit4: XawCommand {
   Attributes:
      parent: main_background;
      horizDistance: 20;
      vertDistance: 170;
      height: 50;
      width: 50;
      label: 4;
   Methods:
      notify:{
        display := 10 * display + 4;
      }
}
/*
** This command button causes the display to be reset
** to 0.
*/



clear_display: XawCommand {
   Attributes:
```

```
        parent: main_background;
        label: "CE";
        horizDistance: 20;
        vertDistance: 310;
        height: 50;
        width: 50;
    Methods:
        notify: {
          display := 0;
        }
}
```

>



Calculator – Invoking Methods

Example 3: Creating a Menu Bar

This example demonstrates the use of Slang to produce a menu bar with two submenus. The top menu bar presents three options: **Menu_1**, **Menu_2**, and **Quit**. When the user chooses **Menu_1** another menu is displayed. The user can select **Item 2** to display a submenu. If the user chooses **Close** on either submenu only that submenu will close, illustrating the creation of tear off menus. Choosing **Quit** will terminate the program.

```
#include "sat.ill"

|||

/*
** This demonstrates the use of Slang to produce
** a menu bar with two tear off menus.
** Initially, there is a menu bar presented to
** the user with two options: Menu_1 and Menu_2.
** Only Menu_1 is active. When the user selects
** Menu_1, a pull down menu will be displayed
** with additional items. When the user selects
** "Item 2 ->" another menu will be displayed.
** Each pull down has its own "Close" button and
** only affects that pull down menu. When the
** user selects the "Close" from the first pull
** down menu, the other pull down menu will
** remain on the display.
*/

Variables:

 display_menu1_submenu : FALSE;
 display_menu2_submenu : FALSE;
 display_sub_item_submenu : FALSE;

Objects:

menu_bar_form: XawBboard {
```

```
    Attributes:
        height:250;
        width: 250;
}

menu_bar: XawBboard {
    Attributes:
        parent: menu_bar_form;
        height:200;
        width: 200;
        vertDistance:20;
        horizDistance:20;
        borderWidth: 3;
}

menu1_item: XawCommand {
    Attributes:
        parent: menu_bar;
        vertDistance: 10;
        horizDistance: 10;
        height: 20;
        width: 50;
        label: "Menu_1";

    Methods:
        notify: {
          display_menu1_submenu := TRUE;
        }
}

menu2_item: XawCommand {
    Attributes:
        parent: menu_bar;
        vertDistance: 10;
        horizDistance: 60;
        height: 20;
        width: 50;
        label: "Menu_2";

    Methods:
        notify: {
            display_menu2_submenu := TRUE;
```

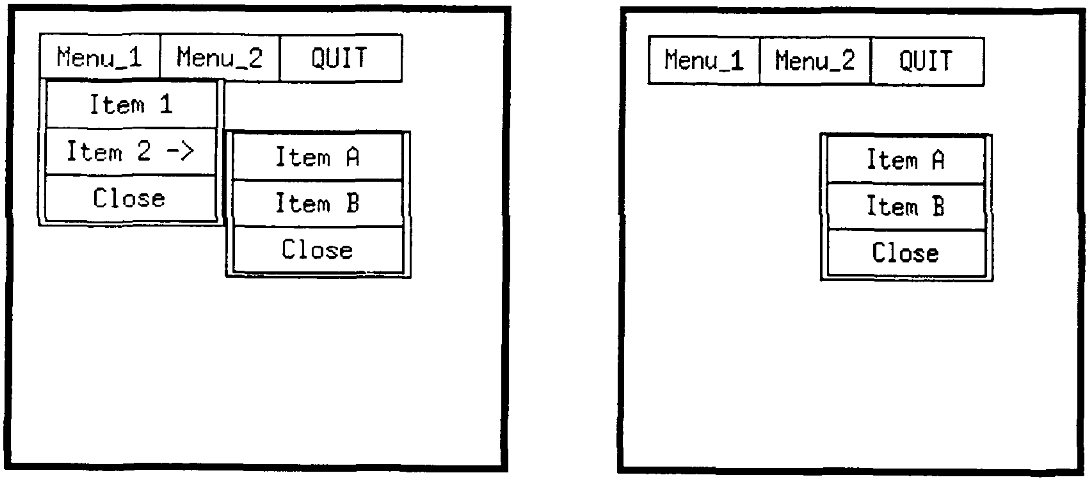
```
        }
    }

    quit_menu_item: XawCommand {
     Attributes:
        parent: menu_bar;
        height: 20;
        width: 50;
        vertDistance: 10;
        horizDistance: 110;
        label: "QUIT";

     Methods:
        notify: {
          exit ();
        }
    }
    /*
    ** menu1 view controller
    */

    VC: menu1_submenu

    Creation Condition: (display_menu1_submenu)

    Objects:

    menu1_form: XawBboard {
     Attributes:
        parent: menu_bar;
        vertDistance: 30;
        horizDistance:10;
        borderWidth: 1;
        height: 65;
        width: 76;
    }

    item1_menu_item: XawCommand {
     Attributes:
        parent: menu1_form;
        vertDistance: 0;
        horizDistance: 2;
```

```
    height: 20;
    width: 70;
    borderWidth: 1;
    label: "Item 1";
}

item2_menu_item: XawCommand {
 Attributes:
    parent: menu1_form;
    vertDistance: 21;
    horizDistance: 2;
    height: 20;
    width: 70;
    borderWidth: 1;
    label: "Item 2 ->";
 Methods:
    notify: {
       display_sub_item_submenu := TRUE;
    }
}

remove_menu_item: XawCommand   {
 Attributes:
    parent: menu1_form;
    vertDistance: 42;
    horizDistance: 2;
    borderWidth: 1;
    height: 20;
    width: 70;
    label: "Close";
 Methods:
    notify:   {
      display_menu1_submenu := FALSE;
    }
}

ENDVC menu1_submenu


/*
** sub_item_submenu view controller
*/
```

Example Programs

```
VC: sub_item_submenu

Creation Condition: (display_sub_item_submenu)

Objects:

sub_item_form: XawBboard {
 Attributes:
    parent: menu_bar;
    vertDistance: 53;
    horizDistance: 88;
    borderWidth: 1;
    height: 65;
    width: 76;
}

itema_menu_item: XawCommand {
 Attributes:
    parent: sub_item_form;
    vertDistance: 0;
    horizDistance: 2;
    height: 20;
    width: 70;
    borderWidth: 1;
    label: "Item A";
}

itemb_menu_item: XawCommand {
 Attributes:
    parent: sub_item_form;
    vertDistance: 21;
    horizDistance: 2;
    height: 20;
    width: 70;
    borderWidth: 1;
    label: "Item B";
}
remove_menu_item: XawCommand    {
 Attributes:
    parent: sub_item_form;
    vertDistance: 42;
```

```
        horizDistance: 2;
        borderWidth: 1;
        height: 20;
        width: 70;
        label: "Close";
    Methods:
        notify: {
          display_sub_item_submenu := FALSE;
        }
}


ENDVC sub_item_submenu
```



Menu Bar

Example 4: Combining Serpent Concepts

This example, although complicated, is intended to show the power of combining the concepts of application shared data, view controllers as logical groupings of objects, and the automatic propagation of dependencies through both local variables and application shared data.

The Slang program demonstrates a simple prototyper for command widgets. It allows the user to create command widgets and manipulate them based on textual input into a table. There are several different logical functions in the example:

- Present a canvas that will be filled in by the created display. (A drawing surface if the example used a direct manipulation interface.)
- Present a button to use to create new elements for the canvas.
- Present a table that displays and allows modification of the current attributes of each element.
- Present the created element itself.

The form named `main_background` in the example acts as the canvas and the `command_widget` named `create` presents a button to be used to create a new instance of a command widget for the end user to manipulate. These are elements seen in the previous examples.

New elements are added to the example beginning with the method for the `create_object`. This method adds a new instance of the command element of application shared data. It does this using the `create_sd_instance` function. This function is used to create a new instance of application shared data.

The view controller template `command_instance` has a creation condition that is watching for new command element instances. The creation condition of a view controller template creates a new instance of the view controller whenever the condition is true. Thus, the creation of a new instance of application shared data will trigger the creation of another instance of the view controller.

When the view controller is created, a table that displays and gives the end user the opportunity to modify the attributes of the created command button is also created. This table is based on the `att_background` form and has a number of elements that will be discussed. Also created at the same time is the `command_widget` named `command_real` that is placed on the canvas and that gives a visual presentation of the attributes. Whenever the end user selects the create button, a new `command_real` widget and its attributes are created and displayed.

The total sequence of events to place a widget on the canvas is:

1. The end user selects the command widget `create`.
2. The method for `create` is activated and a new element is added to the command record in application shared data.
3. A new instance of the `command_instance` view controller is created together with its associated objects.
4. The objects created are:
    - att_background
    - the ten objects that display the attributes:
        - `att_text_label`
        - `att_text_value`
        - `att_height_label`
        - `att_height_value`
        - `att_width_label`
        - `att_width_value`
        - `x label`
        - `x_value`
        - `y_label`
        - `y_value`
        - `att_vert_value`
    - the displayed command widget itself, `command_real`

To understand how modification of the attributes results in modification of the displayed widget, the structure of the table of attributes must be understood. Each entry in the table consists of a label widget and a text widget. The label widget gives a description of what the value of the text widget represents. All of these widgets are based on the `att_background` form. There is also a command button labeled "ok" on the form. The end user modifies those attributes desired and then selects the "ok" button. Each text widget has an attribute named `send_buffer` that tells the widget whether to report the current value of its text. The widget holds its text unless `send_buffer` is 1.

In the table, send_buffer is made to depend upon a local variable named send_flag. (This is another example of the propagation of dependencies.) The variable send_flag is initially set to 0 and the send_buffer attribute of each of the text buffers is set to send_flag. Thus, the send_buffer attribute of each of the text buffers is initially set to ask the text buffer to hold its text. When the end user selects the "ok" button, send_flag is set to 1. This automatically sets the send_buffer of each text widget to 1 and asks the text widget to send its value. This value is placed in text_buffer and the send method is generated for the text_widget. Each text widget's method sets the correct value of the displayed widget into application shared data. This, in turn, is propagated to the displayed copy of the widget on the canvas.

If there are multiple instances of the displayed widget, each is owned by a particular instance of the command_instance view controller, and each instance of that view controller is bound to an instance of the command record in application shared data. Consequently, the attributes displayed to the end user are associated with a command widget on the canvas.

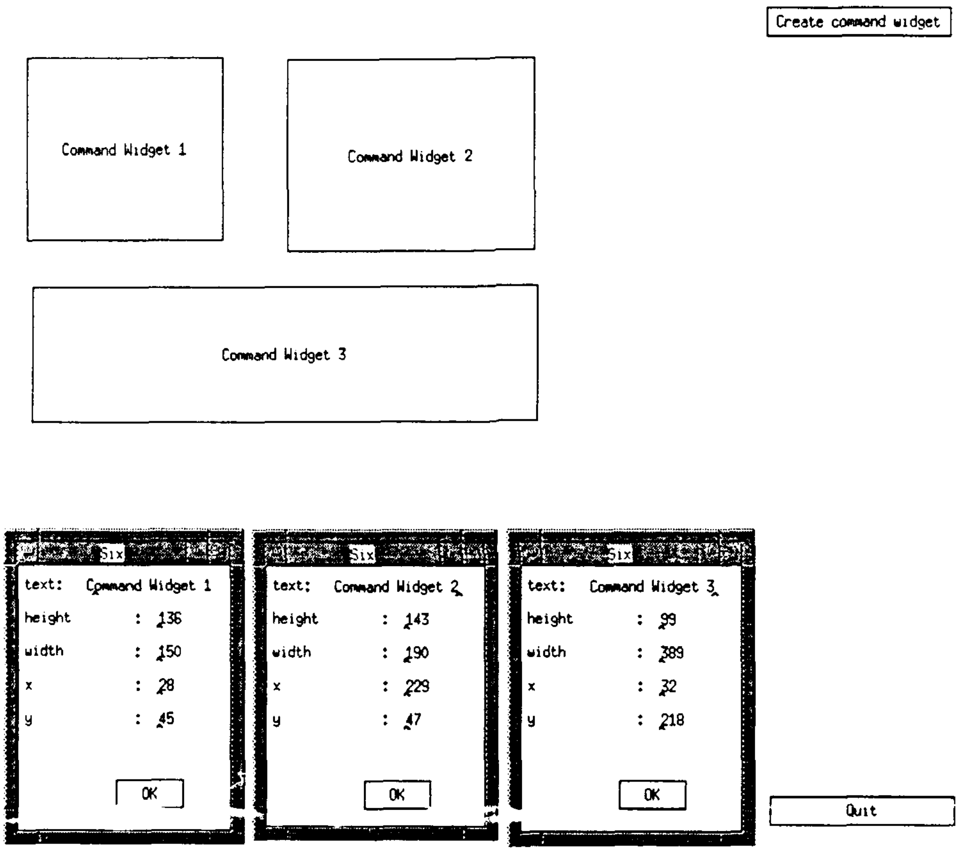
Shared Data Definition File

The shared data definition used in this example defines the communication between the dialogue and the application, which saves the values of the prototyping. The attributes of the created command widgets are passed to the prototyper.

```
<<<    >>>
app: shared data
command: record
            text:      string[32];!included text
            vert_distance:integer;!y position
            horiz_distance:integer;!x position
            width:     integer; !width of widget
            height:    integer; !height of widget
end record;
end shared data;
```

Slang Code

This Slang program demonstrates a simple prototyper for command widgets. It allows the end user to create command widgets and manipulate them based on textual input into a table.



Prototyper for Command Widgets

Initially created are a form upon which the command widgets are positioned and a button which when pressed will create another command widget.

Example Programs

There is a view controller template which is instantiated for each command widget created. It controls the input of the attributes and the assigning of those attributes to the command widgets displayed for modification.

This slang program is not very general. That is, the addition of something beside a command widget would require essentially replicating the entire program. It is intended as an example to demonstrate the use of Serpent with an application.

```
#include "sat.ill"
#include "dm.ill"

|||
/*
** This is the slang program to demonstrate a simple
** prototyper for command widgets. It allows the end
** user to create command widgets and manipulate them
** based on textual input into a table.

** Initially created are a form upon which the command
** widgets are positioned and a button which when
** pressed will create a command widget.

** There is a view controller template which is
** instantiated for each command widget created. It
** controls the input of the attributes and the
** assigning of those attributes to the command widgets
** displayed for modification.

** This slang program is not very general. That is, the
** addition of something beside a command widget would
** require essentially replicating the entire program.
** It is intended as an example to demonstrate the use
** of Serpent with an application.
*/

/*Two objects are defined here.
** 1. a form for the background of all of the objects
** 2. a button to create a new command widget for
** management
*/
```

```
Objects:

main_background: XawBboard {
 Attributes:
    width: 765;
    height: 645;
 }

create_btn: XawCommand {
 Attributes:
    parent: main_background;
    label: "Create command widget";
    width: 140;
    height: 20;
    vertDistance: 10;
    horizDistance: 600;
 Methods:
    notify: {
      create_element("command", "DM_BOX");
    }
}

quit_widget: XawCommand {
 Attributes:
    parent: main_background;
    label: "Quit";
    width: 140;
    height: 20;
    vertDistance: 600;
    horizDistance: 600;
 Methods:
     notify: {
        exit();
     }
}

/*
** The "new" function will create a view controller
** instance whenever a new instance of the command
** element is added to shared data.
*/
```

```
VC: command_instance

Creation Condition: (new("command"))

/*
** The components being set represent those variables
** which can be modified through the attribute form.
** The settings are default values to be modified. The
** modifications are automatically displayed in the
** realization window.
*/
Variables:

   send_flag : 0;
   local_variable : 1;
   command_x : 0;
   command_y : 0;
   command_w : 100;
   command_h : 20;

Objects:

/*
** The following objects form a table which allows the
** end user to input attributes textually
*/

att_background: XawBboard {
 Attributes:
    width: 200;
    height: 200;
 }

/*
** The ok_button signals the text widgets with their
** attributes to send their current values.
*/

ok_button: XawCommand {
 Attributes:
    parent: att_background;
```

```
        justify: 1;/* center */
        label: "OK";
        width: 50;
        height: 20;
        fromHoriz: NULL;
        horizDistance: 75;
        fromVert: NULL;
        vertDistance: 160;
    Methods:
        notify: {
          send_flag := 1;
        }
}

/*
** The next two objects represent an entry in the table
** which provides the text: "text" and the value. This
** allows modification of the text within the
** constructed widget
*/

att_text_label: XawLabel {
 Attributes:
        parent: att_background;
        justify: 0;/* left */
        label: "text:";
        fromHoriz: NULL;
        fromVert: NULL;
        borderColor: "white";
        vertDistance: 0;
        horizDistance: 0;
        height: 25;
        width: 50;
}

att_text_value: XawText {
 Attributes:
        parent: att_background;
        width: 175;
        height: 25;
        fromHoriz: NULL;
        fromVert: NULL;
```

```
    borderColor: "white";
    vertDistance: 5;
    horizDistance: 50;
    editType: 2;
    sendBuffer: send_flag;
 /* value of 1 is message to send text*/
    textBuffer: "<label>";
 Methods:
    send: {
      command.text := textBuffer;
      send_flag := 0;
    }.
}

/*
** The next two objects represent an entry in the table
** which provides the text: "height" and the value. This
** allows modification of the size of the constructed
** widget
*/

att_height_label: XawLabel {
 Attributes:
    parent: att_background;
    justify: 0;
    label: "height        :";
    fromHoriz: NULL;
    fromVert: NULL;
    borderColor: "white";
    vertDistance: 25;
    horizDistance: 0;
    height: 25;
    width: 105;
}

att_height_value: XawText {
 Attributes:
    parent: att_background;
    width: 120;
    height: 25;
    fromHoriz: NULL;
    fromVert: NULL;
```

```
        borderColor: "white";
        vertDistance: 30;
        horizDistance: 105;
        editType: 2;
        sendBuffer: send_flag;
       /* value of 1 is message to send text*/
        textBuffer: command_h ;
    Methods:
        send: {
          commandheight := textBuffer;
          send_flag := 0;
        }
}

/*
** The next two objects represent an entry in the table
** which provides the text: "width" and the value. This
** allows modification of the size of the constructed
** widget
*/

att_width_label: XawLabel {
    Attributes:
        parent: att_background;
        justify: 0;
        label: "width          :";
        fromHoriz: NULL;
        fromVert: NULL;
        borderColor: "white";
        vertDistance: 50;
        horizDistance: 0;
        height: 25;
        width: 105;
}

att_width_value: XawText {
    Attributes:
        parent: att_background;
        width: 120;
        height: 25;
        fromHoriz: NULL;
        fromVert: NULL;
```

```
        borderColor: "white";
        vertDistance: 55;
        horizDistance: 105;
        editType: 2;
        sendBuffer: send_flag;
        textBuffer: command_w;
     Methods:
        send: {
          command.width := textBuffer;
          send_flag := 0;
        }
    }

    /*
    ** The next two objects represent an entry in the table
    ** which provides the text: "horizDistance" and the
    ** value. This allows modification of the position of
    ** the constructed widget
    */

    att_horiz_label: XawLabel {
     Attributes:
        parent: att_background;
        justify: 0;
        label: "x             :";
        fromHoriz: NULL;
        fromVert: NULL;
        borderColor: "white";
        vertDistance: 75;
        horizDistance: 0;
        height: 25;
        width: 105;
    }

    att_horiz_value: XawText {
     Attributes:
        parent: att_background;
        width: 120;
        height: 25;
        fromHoriz: NULL;
        fromVert: NULL;
        borderColor: "white";
```

```
    vertDistance: 80;
    horizDistance: 105;
    editType: 2;
    sendBuffer: send_flag;
    textBuffer: command_x;
    length: 10 ;
 Methods:
    send: {
      commandhoriz_distance := textBuffer;
      send_flag := 0;
    }
}

/*
** The next two objects represent an entry in the table
** which provides the text: "vertDistance" and the
** value. This allows modification of the position of
** the constructed widget
*/

att_vert_label: XawLabel {
 Attributes:
    parent: att_background;
    justify: 0;
    label: "y             :";
    fromHoriz: NULL;
    fromVert: NULL;
    borderColor: "white";
    vertDistance: 100;
    horizDistance : 0;
    height: 25;
    width: 105;
}

att_vert_value: XawText {
 Attributes:
    parent: att_background;
    width: 120;
    height: 25;
    fromHoriz: NULL;
    fromVert: NULL;
    borderColor: "white";
```

```
        vertDistance: 105;
        horizDistance: 105;
        editType: 2;
        sendBuffer: send_flag;
        textBuffer: command_y;
    Methods:
        send: {
          command.vert_distance := textBuffer;
          send_flag := 0;
        }
}

/*
** The following object actually displays the
** widget which is created and which has
** attributes specified from the table
*/

 VC: XawCommand_instance

 Creation Condition: (local_variable = 1)

 Objects:

 command_real: XawCommand {
    Attributes:
      parent: main_background;
      label: att_text_value.textBuffer;
      width: att_width_value.textBuffer;
      height: att_height_value.textBuffer;
      justify: 1;/* center */
      vertDistance: att_vert_value.textBuffer;
      horizDistance: att_horiz_value.textBuffer;
      x: horizDistance;
      y: vertDistance;
      allowUserResize: TRUE;
      allowUserMove: TRUE;
    Methods:
      move: {
        command_x := x;
        command_y := y;
      }
```

```
        resize: {
          command_x := x;
          command_y := y;
          command_w := width;
          command_h := height;
        }
  }
    ENDVC XawCommand_instance

ENDVC command_instance
```

Example Programs

Index

UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION
Unclassified

1b. RESTRICTIVE MARKINGS
None

2a. SECURITY CLASSIFICATION AUTHORITY
N/A

2b. DECLASSIFICATION/DOWNGRADING SCHEDULE
N/A

3. DISTRIBUTION/AVAILABILITY OF REPORT
Approved for Public Release
Distribution Unlimited

4. PERFORMING ORGANIZATION REPORT NUMBER(S
CMU/SEI-91-UG-1

5. MONITORING ORGANIZATION REPORT NUMBER(S)
CMU/SEI-91-UG-1

6a. NAME OF PERFORMING ORGANIZATION
Software Engineering Institute

6b. OFFICE SYMBOL (if applicable)
SEI

7a. NAME OF MONITORING ORGANIZATION
SEI Joint Program Office

6c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh PA 15213

7b. ADDRESS (City, State and ZIP Code)
ESD/AVS
Hanscom Air Force Base, MA 01731

8a. NAME OFFUNDING/SPONSORING ORGANIZATION
SEI Joint Program Office

8b. OFFICE SYMBOL (if applicable)
ESD/AVS

9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
F1962890C0003

8c. ADDRESS (City, State and ZIP Code)
Carnegie Mellon University
Pittsburgh PA 15213

10. SOURCE OF FUNDING NOS.

PROGRAM ELEMENT NO	PROJECT NO.	TASK NO	WORK UNIT NO.
63752F	N/A	N/A	N/A

11. TITLE (Include Security Classification)
Serpent: Overview

12. PERSONAL AUTHOR(S)
SEI User Interface Project

13a. TYPE OF REPORT
Final

13b. TIME COVERED
FROM TO

14. DATE OF REPORT (Yr., Mo., Day)
April 1991

15. PAGE COUNT

16. SUPPLEMENTARY NOTATION

17. COSATI CODES

FIELD	GROUP	SUB. GR.

18. SUBJECT TERMS (Continue on reverse of necessary and identify by block number)
Serpent, UIMS, user interface management system, user interface generators

19. ABSTRACT (Continue on reverse if necessary and identify by block number)
Serpent is a user interface management system (UIMS) that supports the development and implementation of user interfaces, providing an editor to specify the user interface and a runtime system that enables communication between the application aand the end user. This document provides an overview of the Serpent system. It is intended for software engineers involved in user interface development and assumes no previous knowledge of Serpent.

(please turn over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT
UNCLASSIFIED/UNLIMITED ■ME AS RPTDTIC US□S ■

21. ABSTRACT SECURITY CLASSIFICATION
Unclassified, Unlimited Distribution

22a. NAME OF RESPONSIBLE INDIVIDUAL
John S. Herman, Capt, USAF

22b. TELEPHONE NUMBER (Include Area Code)
(412) 268-7630

22c. OFFICE SYMBOL
ESD/AVS (SEI

DD FORM 1473, 83 APR EDITION of 1 JAN 73 IS OBSOLETE UNLIMITED, UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS

ABSTRACT —continued from page one, block 19